MS#160316.1/M&G#40062.101-US-01

METHOD AND SYSTEM FOR TRACING

Technical Field

The present invention relates to application development and more particularly to collecting and outputting runtime information for software applications running on a computing system.

Background

Developing software applications can be difficult. Typically, the development process results in errors during compilation that must be debugged before the final program is complete. Debugging these errors can be complex and difficult, especially as the software application becomes more complex. Inadequate debugging tools and/or weak instrumentation data often compound this difficulty. In addition, complex applications demand optimization to conserve memory, processing time, or other overhead considerations. Therefore, improvements are desirable.

Summary

In accordance with the present invention, the above and other problems are solved by the following:

In one aspect of the present invention, a method of collecting runtime information for an application in a computing system is described. The method

includes locating trace statements in a source code of the application, collecting information regarding the trace statements, and outputting the information for use by a user.

In another aspect of the present invention, a system for collecting runtime information for an application in a computing system is described. The system includes a locate module, a collect module, and an output module. The locate module locates trace statements in a source code of an application. The collect module collects information regarding the trace statements. The output module outputs the information for use by a user.

The invention may be implemented as a computer process, a computing system, or as an article of manufacture such as a computer program product. The computer program product may be a computer storage medium readable by a computer system and encoding a computer program of instructions for executing a computer process. The computer program product may also be a propagated signal on a carrier readable by a computing system and encoding a computer program of instructions for executing a computer process.

A more complete appreciation of the present invention and its scope may be obtained from the accompanying drawings, that are briefly described below, from the following detailed descriptions of presently preferred embodiments of the invention and from the appended claims.

Brief Description of the Drawings

Referring now to the drawings in which like reference numbers represent corresponding parts throughout:

Fig. 1 is a schematic representation of methods and systems for a trace system, according to an exemplary embodiment of the present invention;

Fig. 2 is a schematic representation of a computing system that may be used to implement aspects of the present invention;

Fig. 3 is a schematic representation of a server computing arrangement that may be used to implement aspects of the present invention; and

Fig. 4 is a flow chart illustrating the logical operations of a tracing system that may be used to implement aspects of the present invention.

Detailed Description

In the following description of preferred embodiments of the present invention, reference is made to the accompanying drawings that form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention.

In general, the present disclosure describes methods, systems, and an article of manufacture containing the methods for tracing in a software application on a computing system. Trace statements can be inserted throughout the source code. The

trace function can be enabled or disabled. When enabled, the trace system will track information about the trace statements, cookie information, header information, form variables, query strings, application state objects, session state objects, control tree, and server variables. When disabled, the trace statements are ignored at run time. The collected information can be output to the bottom of the page rendered or to a separate application.

This type of debugging structure provides cross-language debugging support that can be used both locally and remotely from a web server. The trace capability enables both controls and page developers to append instrumentation messages through their source code. These instrumentation messages can be used to simplify the process of identifying what occurs during a given web request and help track down resulting problems.

This process has numerous advantages. One such advantage is that the tracing function provides a secure, accessible means for collecting and viewing runtime information about an application that developers can use to monitor request execution. Another such advantage is that this function provides a general mechanism for outputting debug statements during request execution, to a trace viewer application or to the browser window.

Referring now to **Fig. 1**, a schematic representation of a trace system **100** is illustrated. At block **105**, the trace system **100** collects runtime information. This information can include, for example, when did a runtime request occur, what was the

status code, and what was the request type. At block **110**, the trace system **100** outputs the runtime information, for example, for viewing. This information can be output to a trace viewer application or to a browser window. Thus a developer or administrator can monitor the execution of an application, noting such items as time and resources used. In addition, a developer can find errors and debug the application.

Fig. 2 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention might be implemented. Although not required, the invention is described in the general context of computer-executable instructions, such as program modules, being executed by a computing system. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types.

Those skilled in the art will appreciate that the invention might be practiced with other computer system configurations, including handheld devices, palm devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, network personal computers, minicomputers, mainframe computers, and the like. The invention might also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules might be located in both local and remote memory storage devices.

Referring now to **Fig. 2**, an exemplary environment for implementing embodiments of the present invention includes a general purpose-computing device in the form of a computing system **200**, including at least one processing system **202**. A variety of processing units are available from a variety of manufacturers, for example, Intel or Advanced Micro Devices. The computing system **200** also includes a system memory **204**, and a system bus **206** that couples various system components including the system memory **204** to the processing unit **202**. The system bus **206** might be any of several types of bus structures including a memory bus, or memory controller; a peripheral bus; and a local bus using any of a variety of bus architectures.

Preferably, the system memory **204** includes read only memory (ROM) **208** and random access memory (RAM) **210**. A basic input/output system **212** (BIOS), containing the basic routines that help transfer information between elements within the computing system **200**, such as during start-up, is typically stored in the ROM **208**.

Preferably, the computing system **200** further includes a secondary storage device **213**, such as a hard disk drive, for reading from and writing to a hard disk (not shown), and a compact flash card **214**.

The hard disk drive **213** and compact flash card **214** are connected to the system bus **206** by a hard disk drive interface **220** and a compact flash card interface **222**, respectively. The drives and cards and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for the computing system **200**.

Although the exemplary environment described herein employs a hard disk drive **213** and a compact flash card **214**, it should be appreciated by those skilled in the art that other types of computer-readable media, capable of storing data, can be used in the exemplary system. Examples of these other types of computer-readable mediums include magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, CD ROMS, DVD ROMS, random access memories (RAMs), read only memories (ROMs), and the like.

A number of program modules may be stored on the hard disk **213**, compact flash card **214**, ROM **208**, or RAM **210**, including an operating system **226**, one or more application programs **228**, other program modules **230**, and program data **232**. A user may enter commands and information into the computing system **200** through an input device **234**. Examples of input devices might include a keyboard, mouse, microphone, joystick, game pad, satellite dish, scanner, and a telephone. These and other input devices are often connected to the processing unit **202** through an interface **240** that is coupled to the system bus **206**. These input devices also might be connected by any number of interfaces, such as a parallel port, serial port, game port, or a universal serial bus (USB). A display device **242**, such as a monitor, is also connected to the system bus **206** via an interface, such as a video adapter **244**. The display device **242** might be internal or external. In addition to the display device **242**, computing systems, in general, typically include other peripheral devices (not shown), such as speakers, printers, and palm devices.

When used in a LAN networking environment, the computing system **200** is connected to the local network through a network interface or adapter **252**. When used in a WAN networking environment, such as the Internet, the computing system **200** typically includes a modem **254** or other means, such as a direct connection, for establishing communications over the wide area network. The modem **254**, which can be internal or external, is connected to the system bus **206** via the interface **240**. In a networked environment, program modules depicted relative to the computing system **200**, or portions thereof, may be stored in a remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computing systems may be used.

The computing system **200** might also include a recorder **260** connected to the memory **204**. The recorder **260** includes a microphone for receiving sound input and is in communication with the memory **204** for buffering and storing the sound input. Preferably, the recorder **260** also includes a record button **261** for activating the microphone and communicating the sound input to the memory **204**.

A computing device, such as computing system **200**, typically includes at least some form of computer-readable media. Computer readable media can be any available media that can be accessed by the computing system **200**. By way of example, and not limitation, computer-readable media might comprise computer storage media and communication media.

Computer storage media includes volatile and nonvolatile, removable and non-

removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium that can be used to store the desired information and that can be accessed by the computing system **200**.

Communication media typically embodies computer-readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media includes wired media such as a wired network or direct-wired connection, and wireless media such as acoustic, RF, infrared, and other wireless media. Combinations of any of the above should also be included within the scope of computer-readable media. Computer-readable media may also be referred to as computer program product.

Referring now to **Fig. 3**, an exemplary environment **300** for implementing embodiments of the present invention includes a multi-tier architecture comprising a client tier **302**, a server tier **304**, and a database tier **306**. This multi-tier architecture is common, for example, in applications related to the World Wide Web or Internet. The

client tier **302**, the server tier **304**, and the database tier **306** might contain any number of computing systems, for example, the computing system **200** as described in connection with **Fig. 2**. In the embodiment illustrated in **Fig. 3**, the client tier **302** includes a plurality of client computing systems **308**, **310**, **312**, **314**. Likewise, the server tier **304** includes a plurality of server computing systems **316**, **318**, **320**, **322**, **324**. The database tier **306** includes a database **326**. Of course, the database tier **306** could also contain multiple databases. This exemplary environment is commonly referred to as a three-tier architecture.

In this three-tier architecture, the database tier **306** is the ultimate repository of information or data. The client tier **302** provides the user-interface logic for interfacing with a user of the three-tier architecture. The server tier **304** typically contains the business rules and controls the interface between the client tier **302** and the database tier **306**. The client tier **302** typically interacts with the server tier **304**, which in turn interacts with the database tier **306** to update the database **326** and retrieve data from the database.

Many users can interface with the server tier **304** simultaneously through a router **328**. As stated previously, the client tier **302** might contain several, thousands, or millions of client computing systems **308**, **310**, **312**, **314**, for example a web browser or a rich client. All of these computing systems **308**, **310**, **312**, **314** interact with the router **328** sending requests to the server tier **304** for information and receiving information back from the server tier **304**. For example, the client computing system **308** might

request a list of books for sale from the server tier **304**. The client computing system **308** sends the request to the router **328**. The router **328** routes the request to a server computing system, such as the server computing system **316**, for example a web server or network server. The router **328** might be a dumb router or a smart router as is commonly known. Preferably, the router **328** routes the client requests based on load. In other words, the router **328** determines which server computing system **316**, **318**, **320**, **322**, **324** in the server tier **304** has the most capacity available at the time of the request and accordingly routes the request.

The server computing system **316** receives the request from the client computing system **308**, through the router **328**, and accesses the database **326** to retrieve the requested information. The database **326** sends the requested information to the server computing system **316**. The server computing system **316** processes or formats the information and returns the resulting information to the client computing system **308** through the router **328** for use by the user. As the number of requests to the server tier **304** increases, more server computing systems can be added to the server tier **304**. Thus, the multi-tier architecture is scalable.

Figure 4 is a flow chart illustrating the logical operations of a tracing system **400**. Operation flow begins at a start position **402.** A detect module **404** detects if a tracing function is enabled. Preferably, the tracing function is enabled by a user, such as a developer or administrator, by turning the tracing function on within the source code itself.

Typically, by default the trace function is not enabled on a page, such that the "Trace.IsEnabled" variable will return false. It is noted that the trace function can be enabled and disabled at any point within the source code. Thus, if a user wanted trace information for only a portion of the application, the above syntax can be inserted at the point in the source code that the user wants to begin tracing. The following syntax can be inserted at the point in the source code that the user wants to stop tracing:

"Trace.IsEnabled = false"

If the detect module **404** detects that the trace function is not enabled, operational flow branches "NO" to the ignore operation **406**. The ignore operation **406** ignores the trace statements and no messages will be output during a web request. Ignoring the trace statements saves system performance. Operational flow proceeds to termination point **408** where operational flow ends.

If the detect module **404** detects that the trace function is enabled, or, in other words, that a true value is returned for the "Trace.IsEnabled" variable, operational flow branches "YES" to a locate operation **410**. The locate operation **410** locates the trace statements within the source code that occurred during a web request. Preferably, these trace statements are placed in the source code by a user. The trace property is exposed in the HttpContext for a request. The trace property is also available directly from the Page class. The trace statements can be organized using a "category" string. For example, the following is a sample page containing a trace statement:

```
<html>

    <script runat=server>

        Sub Page_Load(ByVal Sender as Object, ByVal E as EventArgs)
            Trace.Write("Merchant", "Customer State is CA")
        End Sub

    </script>

    <body>

        <form runat=server>
            <span id="Message" runat=server/>
            </form>

        </body>

</html>
```

Preferably, trace messages can be organized by using a "tracemode" attribute exposed on the page directive. A value of "SortByTime" will cause the trace messages to be output in the order in which they were written. A value of "SortByCategory" will cause the trace messages to be output alphabetically by category.

A collect operation **412** collects runtime information for the trace statements. Such runtime information might include when did the request occur; what was the http status code; and what was the request type, i.e., get, or a post request.

A control operation **414** collects server control information, or the tree of controls for a page. For example, the page might contain a form that contains a text box

and a button. The following is an example control using tracing:

```
public class MyButton : Control {
    public override void Render (HTMLTextWriter output) {
        Trace.Write("Render","Button class is about to render....");
        output.Write("<button>Simple Button</button>);
        Trace.Write("Render","Button class has just finished rendering....");
    }
}
```

This server control information allows the developer to see what the application is doing. This is useful in debugging an application. Likewise, a cookie operation **416** will collect cookie information. A header operation **418** will collect header information. A variable operation **420** will collect information about variables.

An output operation **422** will output all of the runtime information, including the information collected by the collect operation **412**, the control operation **414**, the cookie operation **416**, the header operation **418**, and the variable operation **420**, for use by the user. A find operation **424** finds an output configuration. An output module **426** detects whether the output configuration dictates output to a separate application. If the output module **426** determines that the output configuration does not dictate output to a separate application, operational flow branches "NO" to the page operation **428**. The page operation **428** outputs the information collected by the output operation **422** to the bottom of the page being rendered. Preferably, displaying the output to the bottom of

the page is the default display mode. Thus, the Page class will automatically output an HTML table at the conclusion of page rendering that details all of the assorted category trace logs. For example:

```
<html>
    <body>
        <form runat=server>
            <span id="Message" runat=server/>
        </form>
    </body>
</html>

<table>
    <tr>
        <td colspan=2><h1>Trace Log</h1></td>
    </tr>

    <tr>
        <td><h3>Merchant</h3></td>
        <td> Customer State is CA </td>
    </tr>

    <tr>
        <td><h3>Render Category</h3></td>
        <td>
            Button class is about to render.... <br>
            Button class has just finished rendering....
        </td>
    </tr>
</table>
```

Operational flow terminates at termination point **408**.

If the output module **426** determines that the output configuration does dictate output to a separate application, operational flow branches "YES" to a separate operation **430**. The separate operation **430** directs the output to a separate application.

Of course, the tracing system 400 might also check for enablement of output to the bottom of the page as described above after outputting to a separate application. Outputting the runtime information to a separate application has advantages. One such advantage is that the application can be run without the user, who is running the application, seeing the output while a separate user views the output information. Operational flow terminates at termination point **408**.

The operational flow chart depicted in **Fig. 4** may best be understood in terms of application examples. Referring to **Fig 4.**, in a first application example, operational flow begins at start point 402. A developer has inserted numerous trace statements throughout the source code, but has not enabled the trace function. The detect module 404 detects that the trace function is not enabled, and operational flow branches "NO" to the ignore operation 406. The ignore operation 406 ignores the trace statements during compilation, and operational flow ends at termination point 408.

In another application example, the developer has enabled the trace function. The detect module 404 detects that the trace function is enabled, and operational flow branches "YES" to the locate operation 410. The locate operation 410 locates the trace statements. The collection operation 412 collects information for the trace statements. The control operation 414 collect server control operation. The cookie operation 416 collects cookie information. The header operation 418 collects header information. The server operation 420 collects server information.

The output operation 422 outputs the runtime information collected by the collection operation 412, the control operation 414, the cookie operation 416, header operation 418, and the server operation 420. The detect operation 424 detects that the output configuration for outputting the runtime information is to the bottom of the page being rendered. The output module 426 detects that the output is not to a separate application, and operational branches "NO" to the page operation 428. Operational flow ends at termination point 408.

In another application example, operational flow proceeds as described above to the detect operation 424. The detect operation 424 detects that the output configuration dictates outputting to a separate application. The output module 426 detects outputting to a separate application, and operational flow branches "YES" to the separate operation 430. The separate operation 430 outputs the runtime information to a separate application. Operational flow ends at termination point 408.

The logical operations of the various embodiments illustrated herein are implemented (1) as a sequence of computer implemented steps or program modules running on a computing system and/or (2) as interconnected logic circuits or circuit modules within the computing system. The implementation is a matter of choice dependent on the performance requirements of the computing system implementing the invention. Accordingly, the logical operations making up the embodiments of the present invention described herein are referred to variously as operations, steps, engines, or modules.

The various embodiments described above are provided by way of illustration only and should not be construed to limit the invention. Those skilled in the art will readily recognize various modifications and changes that may be made to the present invention without following the example embodiments and applications illustrated and described herein, and without departing from the true spirit and scope of the present invention, which is set forth in the following claims.